Detection of First Order Axiomatic Theories

Guillaume Burel[1], Simon Cruanes[2]

[1] ÉNSIIE/Cédric, 1 square de la résistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr
http://www.ensiie.fr/~guillaume.burel/
[2] École polytechnique and INRIA, 23 Avenue d'Italie, 75013 Paris, France
simon.cruanes@inria.fr
https://who.rocq.inria.fr/Simon.Cruanes/

Abstract Automated theorem provers for first-order logic with equality have become very powerful and useful, thanks to both advanced calculi — such as superposition and its refinements — and mature implementation techniques. Nevertheless, dealing with some axiomatic theories remains a challenge because it gives rise to a search space explosion. Most attempts to deal with this problem have focused on specific theories, like AC (associative commutative symbols) or ACU (AC with neutral element). Even detecting the presence of a theory in a problem is generally solved in an ad-hoc fashion. We present here a generic way of describing and recognizing axiomatic theories in clausal form first-order logic with equality. Subsequently, we show some use cases for it, including a redundancy criterion that can be applied to some equational theories, extending some work that has been done by Avenhaus, Hillenbrand and Löchner.

This is a self-archiving version of the final paper [1], which is available at http://link.springer.com/chapter/10.1007/978-3-642-40885-4_16.

1 Introduction

Automated theorem proving for first order logic has lead to many successful techniques to tackle problems from a lot of application domains. Among the most prominent techniques lies resolution[10]. Superposition[8] appeared later to handle the difficult issue of equality reasoning, that would otherwise drown most provers in a huge search space.

Many theorem provers for first-order logic with equality contain an ad-hoc engine to recognize instances of Associative Commutative (AC) symbols, composed of the two following axioms:

Associativity: $\forall x \forall y \forall z\ x + (y + z) = (x + y) + z$,
Commutativity: $\forall x \forall y\ x + y = y + x$.

[1] DOI: 10.1007/978-3-642-40885-4_16

Once the automated prover has recognized that some symbol has the AC property, it can use some technique to deal with it. However, if similar techniques can be applied to other axiomatic theories — theories that can be defined in terms of a finite set of axioms — code would need to be written for those provers to handle each new theory. We propose here a system that can recognize the presence of theories in a generic and incremental way. The system is based on the use of a second theorem prover, based on Datalog[1], that reasons about the properties that the problem exhibits, rather than trying to solve the problem itself. In some limited sense, this is similar to what a human mathematician does: she would try to use equations and hypotheses on the problem itself, but at the same time she would recognize already met patterns and specific structures (for instance, a group structure, a linear field, or an isomorphism to some other part of the mathematics) and use this meta knowledge to apply theorems and lemmas she knows.

We implemented this technique in our experimental theorem prover Zipperposition. Zipperposition is free software, available under the GPL license at `https://www.rocq.inria.fr/deducteam/Zipperposition/index.html`. It is written in OCaml and implements ordered superposition, with lazy reduction to CNF and automatic selection of a precedence. An embedded Datalog engine is used to reason on properties of the problems, including which known theories and axioms are present; both systems interact by exchanging clauses on the one hand, deduced properties on the other. The superposition prover can use the additional information to infer new clauses thanks to *lemmas* or to activate theory-specific *redundancy criteria*.

Then, we expose two possible applications. The first is a powerful lemma that allows theorem provers that deal well with equality to discover that some relations represent the graph of a function, and to replace instances of the relation by equations. For instance, in the TPTP[13] archive, many algebraic problems on groups (or extensions thereof) are encoded using $\mathrm{sum}(X, Y, Z)$ instead of $Z = \mathrm{add}(X, Y)$. This complicates the axiomatization (many more axioms, that are big Horn clauses, etc.) compared to an equational view of the problem. Our lemma, fed to the prover in a simple declarative language as:

```
functional(r) is axiom ~r(X,Y,Z) | ~r(X,Y,Z2) | Z=Z2.
total(r, f) is axiom r(X,Y,f(X,Y)).
lemma r(X,Y,Z) <=> Z = f(X,Y) if functional(r) and total(r, f).
```

allows to recover an equational (boolean) definition from this encoding, which can then be unfolded to simplify clauses.

The second application is the per-theory activation of an equational redundancy criterion. If we know a saturated, ground convergent system of equations for some theory [2], literals that are tautological or absurd in this theory can be removed while retaining completeness. Our framework allows us to know when such a theory occurs in a problem, so we can use the corresponding redundancy criterion.

We first expose some basic definitions and notations, then successively expose techniques for recognizing individual axioms and whole theories. Then, after

some examples of how to use knowledge about axiomatic theories, we present some experimental results and conclude.

2 Notations and Definitions

The first step toward recognizing theories, is recognizing instances of individual axioms. More complex algebraic theories, like group theory, involve several symbols. We present here a general framework for representing axiom schemas, axiom instances, and for finding instances of the former among the latter. We start with some basic notations.

A *signature* $\Sigma = (S, V)$ is the combination of a finite set of symbols S (with an arity function arity : $S \to \mathbb{N}$), with a countable set of variables V. *Terms* in a signature $\Sigma = (S, V)$ are defined recursively by $t = X \mid f(t_1, \ldots, t_n)$, where $X \in V$ and $f \in S$, with arity$(f) = n$. We give a *type* to each term, in a simple-type system with base types ι for individuals and o for propositions; a function type is written $(\tau_1 \times \cdots \times \tau_n) \to \tau$. The statement "$t$ has type τ" is written $t : \tau$. The set of variables of a term, vars(t), is recursively defined as

$$\mathrm{vars}(X) = \{X\}$$
$$\mathrm{vars}(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \mathrm{vars}(t_i)$$

From now on, f, g, h will be symbols, t, t', t_i will be terms, uppercase letters like X, Y, Z will denote variables and τ will be a type.

A *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$, each *literal* being an equation $s = t$, or the negation of an equation $s \neq t$ (s and t must have the same type). A literal of any sign is written $s \doteq t$. Following [11], we represent propositions p by boolean-typed equations $p = \top$, where $\top : o$ is a special constant for truth.

A *substitution* σ is a finite mapping from variables to terms. The result of applying a substitution to a term t is noted $t\sigma$. If σ and θ are substitutions, then σ is *more general* than θ if there exists η such that $\forall t.\ t\theta = t\sigma\eta$. Usual notions for *most general unifiers* and *most general matcher* are employed:

unifier: The most general unifier of two terms t_1 and t_2, if it exists, is the most general substitution σ such that $t_1\sigma = t_2\sigma$. Terms for which such a substitution exists are said to be *unifiable*.

matcher: The most general matcher of two terms t_1 and t_2, if it exists, is the most general substitution σ such that $t_1\sigma = t_2$. Similarly, it is not always defined.

Equality of two terms modulo a theory E is written $t_1 =_E t_2$, short for $E \vdash t_1 = t_2$. We extend the definitions of unification and matching to *unification modulo AC* and *matching modulo AC*, respectively defined by $t_1\sigma =_{AC} t_2\sigma$ and $t_1\sigma =_{AC} t_2$.

Let the *local signature* of a term or clause, noted $\mathrm{ls}(t)$, be defined as follow:

$$\begin{aligned}\mathrm{ls}(X) &= \emptyset\\ \mathrm{ls}(f(t_1,\ldots,t_n)) &= \{f\} \cup \bigcup_{i=1}^{n} \mathrm{ls}(t_i)\\ \mathrm{ls}(s \doteq t) &= \mathrm{ls}(s) \cup \mathrm{ls}(t)\\ \mathrm{ls}(l_1 \vee \cdots \vee l_n) &= \bigcup_{i=1}^{n} \mathrm{ls}(l_i)\end{aligned}$$

A few special symbols (disjoint from any signature) will be used:

- A *symbol marker*, $\mathfrak{s}$, used to prefix function symbols;
- A *variable marker*, $\mathfrak{v}$, used to prefix variables.

For the meta-prover, we encode properties of the problem at hand in *higher-order logic*, where the definition of terms is extended with *binders* (here, only λ). Well-formed terms for the meta-prover are defined by $t = X \mid f \mid t\ t \mid \lambda X.t$ where $f \in S \cup \{\mathfrak{s}, \mathfrak{v}\}$, $X \in V$ and t a term. Term application $t\ t$ is curried and left-associative. We call *lambda terms* terms in which some variables are bound by a lambda-abstraction. We will assume that the reader knows about basic simply-typed lambda-calculus, but recall that β-reduction is the rule $(\lambda X.t)\ t' \to_\beta [t'/X]t$ and we will work modulo alpha-equivalence in order to prevent variable captures. In the rest of the paper, $t \downarrow^\beta$ denotes the normal form of a term t w.r.t. β-reduction, i.e., the unique term t' such that $t \to_\beta^* t'$ and $\neg\exists t''\ t' \to t''$ (it always exists because simply-typed lambda calculus is convergent).

3 Detecting axioms

The first step toward recognizing theories, is recognizing instances of individual axioms of first-order theories. We will see, in the next section, how to recognize full theories. Many theorem provers contain an ad-hoc system to recognize instances of Associative Commutative (AC) symbols, composed of two axioms:

Associativity: $\forall x \forall y \forall z\ x + (y + z) = (x + y) + z$,
Commutativity: $\forall x \forall y\ x + y = y + x$.

More complex algebraic theories, like group theory, involve several symbols. We present here a general framework for representing axiom schemas, axiom instances, and for finding instances of the former among the latter. Recognizing axioms in any signature is a higher order problem, but we are going to use *currying* to stay in a first-order setting. We start with some basic notations.

We call *pattern* a clause c parametrized by $\mathrm{ls}(c)$. This notion of pattern is central in our approach, since it allows us to reason over axioms and theories regardless of the actual signature of the problem (a given axiom or theory might have several distinct instances within the same proof). A pattern p is represented

as a higher-order *curried* term $t \equiv \lambda X_1 : \tau_1.\lambda X_2 : \tau_2.\ldots\lambda X_n : \tau_n.c$, or more compactly $\Lambda_{i=1}^{n} X_i : \tau_i.c$. We need to curry the term because we cannot replace a function symbol by a variable in first-order terms. c is the *core* of the pattern, and $s_1, \ldots, s_n$ the *input types* or *types* of the pattern. Note that although patterns are higher-order terms (because of the lambda abstractions), we still reason over first-order problems, and any instantiation of a pattern must yield a first-order clause.

Patterns are an extension of the *representative patterns* defined in [4], but are more general because they are curried and deal with non-unit clauses, which explains our use of AC-matching. In addition to that, our technique is concerned with which set of symbols instantiates a given pattern. On the other hand, representative patterns are indexed by an AVL tree, which makes the matching process very efficient.

The point in using curried terms to represent patterns is that we can leverage many well-known techniques, such as AC-matching or term indexing modulo AC[2]. Also, this system is quite easy to adapt to some similar tasks, like matching a pattern $p = \Lambda_i X_i : \tau_i.c$ against a subset of a clause c': we can match $\Lambda_i X_i : \tau_i.(c \vee y)$ against $c' \vee \top$, where $y : o$ is a fresh variable to be matched against the rest of c'. Matching a pattern against a subset of a clause could be useful if the subset is an instance of the negation of the conclusion of a lemma, for instance, because instantiating the lemma would then simplify the clause. The lambda-abstraction is used to have a canonical representation of a pattern (using De Bruijn indexes would also work), so that it can be considered as a constant by Datalog (see section 4). More powerful matching algorithms (e.g., restricted forms of higher-order matching) can be used to find more elaborate instances.

The following property always holds for patterns: if $p = \Lambda_{i=1}^{n} X_i : \tau_i\ c$, and $a_1 : \tau_1, \ldots, a_n : \tau_n$ are terms (in particular, constants), then $p\ a_1\ a_2 \ldots\ a_n$ is a well-typed term, that is isomorphic to a concrete first-order clause.

Pattern abstraction allows us to abstract a clause from its concrete local signature. *Pattern instantiation* applies a pattern to a tuple of symbols, returning a concrete clause. Figure 1 describes the following operations (the variable F used for abstraction is assumed to be a fresh variable uniquely associated with the symbol f):

encoding a term or clause into a curried term, noted $\mathrm{enc}(t)$;
decoding a curried term into a term or clause, noted $\mathrm{dec}(t)$;
abstracting a symbol f out of a curried term t, by a variable F, noted $\mathrm{abs}(t, f, F)$;
applying a curried term t to a term a, noted $\mathrm{app}(t, a)$, extended into the n-ary application $\mathrm{app}(t, a_1, \ldots, a_n)$.

Encoding is a two steps operation: currying, then prefixing variables with $\mathfrak{v}$ and symbols with $\mathfrak{s}$ to force the matching algorithm to bind abstracted symbols (resp. first-order variables) of the pattern only with symbols (resp. variables) of the clause. Decoding is the exact inverse of encoding.

[2] our experimental implementation does not implement AC indexing, though.

$$
\begin{aligned}
\mathrm{enc}(X) &= \mathfrak{v}\ X \\
\mathrm{enc}(f(t_1,\ldots,t_n)) &= \mathfrak{s}\ f\ \mathrm{enc}(t_1)\ \mathrm{enc}(t_2)\ \ldots\ \mathrm{enc}(t_n) \\
\mathrm{enc}(t_1 \doteq t_2) &= \mathrm{enc}(t_1) \doteq \mathrm{enc}(t_2) \\
\mathrm{enc}(l_1 \vee \ldots \vee l_n) &= \mathrm{enc}(l_1) \vee \ldots \vee \mathrm{enc}(l_n) \\
\mathrm{dec}(\mathfrak{v}\ X) &= X \\
\mathrm{dec}(\mathfrak{s}\ f\ t_1\ \ldots\ t_n) &= f(\mathrm{dec}(t_1),\ldots,\mathrm{dec}(t_n)) \\
\mathrm{dec}(t_1 \doteq t_2) &= \mathrm{dec}(t_1) \doteq \mathrm{dec}(t_2) \\
\mathrm{dec}(l_1 \vee \ldots \vee l_n) &= \mathrm{dec}(l_1) \vee \ldots \vee \mathrm{dec}(l_n) \\
\mathrm{abs}(\mathfrak{s}\ f, f, F) &= \mathfrak{s}\ F \\
\mathrm{abs}(\mathfrak{v}\ X, f, F) &= \mathfrak{v}\ X \\
\mathrm{abs}(t_1\ t_2, f, F) &= \mathrm{abs}(t_1, f, F)\ \mathrm{abs}(t_2, f, F) \\
\mathrm{abs}(t_1 \doteq t_2, f, F) &= \mathrm{abs}(t_1, f, F) \doteq \mathrm{abs}(t_2, f, F) \\
\mathrm{abs}(l_1 \vee \ldots \vee l_n, f, F) &= \lambda F.(\mathrm{abs}(l_1, f, F) \vee \ldots \vee \mathrm{abs}(l_n, f, F)) \\
\mathrm{app}(p, a_1, \ldots, a_n) &= (p\ a_1\ a_2\ \ldots\ a_n)\downarrow^{\beta}
\end{aligned}
$$

Figure 1. Rules for patterns

Example: let us consider the theory of commutative monoids ACU with operator f and neutral element e. Its axioms are (last one is *unit*, or U) :

$$
\begin{aligned}
f(X,Y) &= f(Y,X) \\
f(X,f(Y,Z)) &= f(f(X,Y),Z) \\
f(X,e) &= X
\end{aligned}
$$

The corresponding patterns, after currying and abstraction, are:

- $\lambda F.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y) = \mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ X))$
- $\lambda F.(\mathfrak{s}\ F\ (\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y))\ (\mathfrak{v}\ Z) = \mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ Z)))$
- $\lambda F.\lambda E.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{s}\ E) = \mathfrak{v}\ X))$

Once we have a set of patterns $\mathcal{P} = \{p_1,\ldots,p_n\}$, we can *match* those patterns against clauses of the problem we are trying to solve. Matching a pattern p against a clause c amounts to:

1. choose fresh variables $X_1 : s_1, \ldots, X_n : \tau_n$, where $(\tau_i)_i$ are the input types of the pattern p;
2. compute $t \equiv \mathrm{app}(p, X_1, \ldots, X_n)$;
3. use a matching algorithm modulo AC ($=$ is commutative, and $\vee$ is AC) to match t against $\mathrm{enc}(c)$;
4. for each such matcher σ, its restriction $\sigma' \equiv \sigma|_{\{X_1,\ldots,X_n\}}$ is an instance of p equivalent to c (i.e., $\mathrm{dec}(\mathrm{app}(p, X_1\sigma', \ldots, X_n\sigma')) =_{AC} c$).

Example: let us match the pattern for an identity value, $\lambda F.\lambda E.(\mathfrak{s}\ F\ (\mathfrak{s}\ E) = \mathfrak{s}\ E)$ with zero = minus(zero, zero). We first apply the pattern to fresh variables F' and E', obtaining after beta-reduction the HO term $\mathfrak{s}\ F'\ (\mathfrak{s}\ E') = \mathfrak{s}\ E'$. The clause is then encoded into $\mathfrak{s}$ zero = $\mathfrak{s}$ minus ($\mathfrak{s}$ zero) ($\mathfrak{s}$ zero); a solution, obtained by AC-matching the equations, is $\sigma = \{F' \mapsto \text{minus zero}, E' \mapsto \text{zero}\}$. This (first-order) instance can only be found thanks to currying.

The next step is to aggregate several pattern instances into an instance of a *theory*, that is, a set of clauses.

4 Meta-reasoning with Datalog

4.1 Description of an Axiomatic Theory

An *equational theory* is a set of related axioms. Therefore, a *theory pattern* is a set of related clause patterns. We adapt and generalize the mechanism used by Waldmeister[6] for choosing term orderings. In Figure 2, we show a fragment of the file that defines some basic axioms and theories for our prover[3].

```
associative(f) is axiom f(X,f(Y,Z)) = f(f(X,Y), Z).
commutative(f) is axiom f(X,Y) = f(Y,X).
theory ac(f) is associative(f) and commutative(f).
theory aci(f,e) is ac(f) and axiom f(X,e) = X.
```

Figure 2. Description of Theories

This simple file shows us what is needed to define a theory like AC or the theory of commutative monoids. We need to define some axioms, possibly named, to abstract their symbol out, and to constraint symbols of the axioms to be the same. Indeed, the two clauses $f(f(X,Y),Z) = f(X,f(Y,Z))$ and $g(X,Y) = g(Y,X)$ can be matched, respectively, against the axioms `associative(`f`)` and `commutative(`g`)`, but that does not mean that the theory of AC symbols is present. Those axioms are *parametrized* by symbols f and g.

To be able to constraint symbols in the axioms to be the same, we use the *Datalog* fragment of first-order logic[1]. Datalog only allows function-free Horn clauses, but shows very good computational properties, and a set of Datalog clauses always has exactly one minimal model. We are going to have a Datalog reasoner work on properties of the problem, and communicate with the regular superposition prover.

A Datalog atom is of the form $p(t_1, \ldots, t_n)$ where p is a *predicate symbol* and for all i, t_i is either a *Datalog constant* or a *Datalog variable*. A Datalog clause

[3] The axioms, theories, lemmas and redundancy criteria are defined in a file loaded when the theorem prover starts. The format of the file is defined by a simple grammar that is easy to edit and read, as demonstrated in Figure 2.

is a Horn clause, noted a :- $b_1, \dots b_n$. where a is the conclusion, and b_i are the premises We will write a. for unit clauses. We point out that Datalog constants and variables are nothing like the first-order problem's constants and variables. The trick is that the Datalog reasoner will not "see" what is inside the patterns it manipulates (such objects are not expressible in Datalog), but it will consider them as blackboxes (constants).

Let us define the black-boxing of patterns, that embeds first-order objects into Datalog constants. Given a pattern p, we write $\lceil p \rceil$ for the boxed version of p; given such a black box b, we define $\lfloor b \rfloor$ its content, obtained by unboxing. Obviously, $\lfloor \lceil p \rceil \rfloor = p$ must hold. Equality over boxes is defined by $\lceil p \rceil = \lceil q \rceil \Leftrightarrow p = q$. The boxing and unboxing functions are trivially extended to any term.

We can now encode properties about the current problem into Datalog atoms, and their definitions into Datalog clauses. Detecting theories requires a few basic properties, which are the following ones:

- Presence of an instance of a pattern, with the corresponding symbols, such as $\mathrm{app}(p, \mathrm{plus})$ where $p \equiv \lambda F.(\mathfrak{s}\ F\ (\mathfrak{v}\ X)\ (\mathfrak{v}\ Y) = \mathfrak{s}\ F\ (\mathfrak{v}\ Y)\ (\mathfrak{v}\ X))$. It is needed for recognizing the constituting axioms of a theory;
- Presence of a named pattern, for instance `commutative(plus)` (which corresponds to the previous pattern). This is used mainly for the user to attach a meaningful name ("associative") to a pattern;
- Presence of an instance of a theory, with the corresponding symbols, for instance `monoid(plus, zero)`;
- Other properties can be encoded (see Sections 4.4 and 5) for more advanced uses of the Datalog reasoner. This makes the detection mechanism quite generic and modular since it allows to define additional properties based on the previously defined ones.

4.2 Encoding of Properties

Such properties are encoded using a distinct Datalog predicate symbol for each kind of property. This way, new properties can be encoded just by reserving a new predicate symbol for them. The basic properties are encoded by:

pattern: A pattern instance $\mathrm{app}(p, a_1, \dots, a_n)$, is encoded using the predicate "pattern", into $\texttt{pattern}(\lceil p \rceil, \lceil a_1 \rceil, \dots, \lceil a_n \rceil)$

theory: A theory is a name, parametrized by a set of function symbols; A theory instance "name"$(a_1, \dots, a_n)$ is encoded using the predicate "theory" into $\texttt{theory}(\lceil$"name"$\rceil, \lceil a_1 \rceil, \dots, \lceil a_n \rceil)$;

named pattern: It is similar to a theory with a single axiom, but using the predicate "axiom"; so, for instance, $\texttt{associative}(f)$ is encoded into $\texttt{axiom}(\lceil$"associative"$\rceil, \lceil f \rceil)$.

4.3 Encoding of Definitions

It is also necessary to define theories and named patterns, by Datalog clauses that will trigger a property when the constitutive patterns of the theory (respectively

named patterns) are present. This requires Datalog variables; if a theory (named $\mathcal{N}$) is defined by $\mathcal{N}(f_1, \ldots, f_n) \equiv p_1, \ldots, p_m$ (with m premises), its definition will be a Datalog clause with m Datalog atoms as premises. Let us map $f_1, \ldots, f_n$ to fresh Datalog variables $F_1, \ldots, F_n$. The premises p_i are translated to Datalog atoms q_i as follows:

- If p_i expresses the presence of a named pattern or a theory $\mathcal{N}'(f_{\sigma(1)}, \ldots, f_{\sigma(k)})$ parametrized by k symbols, then $q_i = \texttt{axiom}(\lceil \mathcal{N}' \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$ or $q_i = \texttt{theory}(\lceil \mathcal{N}' \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$
- If p_i expresses the presence of a pattern instance $\text{app}(p, f_{\sigma(1)}, \ldots, f_{\sigma(k)})$, with k symbols as parameters, then $q_i = \texttt{pattern}(\lceil p \rceil, F_{\sigma(1)}, \ldots, F_{\sigma(k)})$

The definition is simply $\texttt{theory}(\lceil \mathcal{N} \rceil, F_1, \ldots, F_n)$:- $q_1, \ldots, q_m$. (easily adapted for named patterns).

4.4 Encoding of Other Properties

Other properties, depending on how the prover uses knowledge about theories, can be encoded the same way. For instance, if we want to gather information specifically about AC symbols, we can use a fresh Datalog predicate "ac" and the following clause:

$$\texttt{ac}(F) \text{ :- } \texttt{theory}(\lceil \text{“ac”} \rceil, F)$$

Then, whenever a Datalog fact $ac(a)$ is found by the Datalog reasoner, we know that $\lfloor a \rfloor$ is an associative commutative symbols in the current problem (and we can activate a special strategy to deal with it in the refutational theorem prover, like superposition modulo AC). We will see more detailed examples in Section 5.

4.5 Incremental Computation

When an automated theorem prover tries to solve a given problem, some properties of this problem may not be readily available for recognition. Instead, it may take some time to reach some axioms that are part of a theory's definition. Therefore, *incrementality*, i.e. the ability to discover properties and deduce other properties during the process of solving the problem, is crucial.

Our implementation of the Datalog reasoner therefore does not provide a query interface, but rather an incremental interface; clauses can be added one by one, each time updating the current set of clauses (saturated under the *immediate consequence operator*). The immediate consequence operator adds $a\sigma$ to the set of facts, if a :- $b_1, \ldots, b_n$. is a clause and for all $i \in \{1 \ldots n\}$, $b_i\sigma$ belongs to the set of facts. Because we only work with *safe clauses*, i.e., clauses in which $\text{vars}(a) \subseteq \bigcup_{i=1}^{n} \text{vars}(b_i)$, we are sure that $a\sigma$ is ground.

To achieve incremental computation, the Datalog reasoner is based on unit resolution with selection. Every non-unit clause, of the form a :- $\underline{b_1}, b_2, \ldots, b_n$. with $n > 0$, gets its first body literal *selected* (the underlined literal). Only one inference rule is needed to ensure completeness:

$$\frac{a \text{ :- } \underline{b_1}, b_2, \ldots, b_n. \qquad c. \qquad b_1\sigma = c\sigma}{a\sigma \text{ :- } \underline{b_2\sigma}, b_3\sigma, \ldots, b_n\sigma.}$$

Every time we add a clause to the Datalog reasoner, the resolution rule is applied between clauses of the current fixpoint, and the new clause. Callbacks can be attached to the Datalog reasoner, to be called whenever a new fact is deduced by this inference rule; the new facts are then added to the reasoner one by one – with their own chance to trigger inferences. A non-perfect discrimination tree is used to index selected literals and facts, in order to make this inference reasonably efficient.

4.6 Backward Chaining

In some cases, the underlying first-order calculus used by the theorem prover may never discover some axioms (like associativity). This is the case, for instance, for refutational provers based on resolution or superposition, because they may not need to infer the axiom to remain complete, in case it is deducible from the initial problem but redundant. In this case, if, for instance, out of the 10 axioms that are necessary for an instance of a theory to hold, 9 are present, the Datalog prover may pro-actively spawn a sub-prover to try to show this axiom.

The Datalog reasoner can use prolog-like backward chaining to find which literals may help finding new facts. Assuming we keep a set $\mathcal{G}$ of *goals* – a goal being a literal whose instances may help solving already existing goals – the following rule updates the set of goals:

$$\frac{a \text{ :- } \underline{b_1}, b_2, \ldots, b_n. \qquad g \in \mathcal{G} \qquad a\sigma = g\sigma}{b_1\sigma \in \mathcal{G}}$$

We did not implement a system that spawns sub-provers that attempt to show missing axioms, but it would be quite simple by finding which of the current *goals* of the Datalog reasoner belong to the category of totally instantiated patterns (where all parameters of the pattern are constants, not Datalog variables). However, goals are already important in our implementation, because we only try to match against concrete clauses, the patterns which are currently goals in the Datalog reasoner. In other words, clause patterns we want to match with concrete clauses are $\{p \mid \texttt{pattern}(\lceil p \rceil, F_1, \ldots, F_n) \in \mathcal{G}\}$. The initial set of goals is the set of conclusions of clauses, that is, $\mathcal{G}_0 = \{a \mid a \text{ :- } \underline{b_1}, b_2, \ldots, b_n.\}$, but we could choose a different (restricted) set of goals; for instance, if some lemmas hold only when arithmetic symbols are present, their conclusion shall not be goals until an arithmetic formula is detected, not to clutter the pattern recognition mechanisms.

5 Why Recognize Theories?

In previous sections, we explained how to recognize individual axioms and theories during a saturation proof search. We will now give some ways to use this

knowledge about the problem at hand. Of course, because the coupling with the Datalog reasoner is modular, one can make any use she wants from the output of the Datalog reasoner, and even add whichever clauses and predicates she judges useful. For most uses of the Datalog reasoner, we use a dedicated predicate, and some clauses whose premises are theories or individual axioms. We will call *Knowledge Base* the set of definitions and facts that is given to the theorem prover when it starts; it should contain definitions of axioms, theories, and other data that is specific to how we use knowledge about the problem.

5.1 Lemmas

Let us call *lemma* an already proven logic statement of the form "clause a is true if clauses $b_1, \ldots, b_n$ are". Such a lemma may be a mathematical result that the user makes available to the theorem prover, or some previously proven theorem; all we need to know is that this statement is already known to be proven. It can be encoded in Datalog by abstracting symbols from the conclusion and the premises (see section 4.3).

Our current *Knowledge Base* contains only one lemma that we added by hand, but it has shown to be quite useful. We call this lemma *un-mangling of functional relations*. Given the two properties about a ternary relation symbol r and a binary function symbol f:

- $\texttt{functional}(r) \equiv r(X,Y,Z) \wedge r(X,Y,Z') \Rightarrow Z = Z'$;
- $\texttt{total}(r,f) \equiv r(X,Y,f(X,Y))$.

We know that r encodes the graph of the function f. Hence the following lemma, that states $r(X,Y,Z) \Leftrightarrow (f(X,Y) = Z)$. Its definition is shown in Figure 3. Such a lemma, if applied during the preprocessing phase, allows one to unfold the definition of r, removing it from the problem (our prover uses the calculus of [5], which allows one to use such equivalences for rewriting). After unfolding of r, the problem is "more equational": axioms such as the commutativity of f, that were encoded by $r(X,Y,Z) \wedge r(Y,X,Z') \Rightarrow Z = Z'$, become $f(X,Y) = f(Y,X)$ after simplifications. An equational theorem prover such as E[11] will be able to use more rewriting-based simplification rules.

```
functional(r) is axiom ~r(X,Y,Z) | ~r(X,Y,Z2) | Z=Z2.
total(r, f) is axiom r(X,Y,f(X,Y)).
lemma r(X,Y,Z) <=> Z = f(X,Y) if functional(r) and total(r, f).
```

Figure 3. Un-mangling of Functional Relations Lemma

Before we developed the general theory detection system, specific code was dedicated to recognizing instances of this lemma in Zipperposition. The code took around 85 lines of OCaml and would only work on initial axioms. We

emphasize the fact that detecting instances of this lemma require many features, like detecting non-unit clauses with several abstracted symbols (f and r), and then joining the multi-symbol axioms together. Now, to add similar lemmas, we only need a few lines in the previously mentioned declarative syntax. Each lemma is encoded as exactly one Datalog clause, whose conclusion is a pattern instance.

5.2 Equational Redundancy Criteria

As the authors of [2] point out, superposition-based theorem provers such as SPASS[14], E[11] or Vampire[9] can quickly become overwhelmed by the amount of clauses that are generated in the presence of equational theories such as AC, ACI or other algebraic structures. This is exacerbated by the fact that superposing with commutativity is very often possible in both ways, since the axiom is not oriented by the usual KBO and RPO term orderings. A lot of efforts [12] [3] have been devoted in extending the superposition calculus to work modulo AC, or modulo theories that encompass AC; however it is usually delicate both to implement and prove complete each instance of superposition modulo a theory. We expose here a way of using some knowledge about equational theories to prune the search space of such theorem provers. We will use some definitions and theorems from [2].

Redundancy Criterion Let use consider the section 5 of [2]. A ground convergent system of equations $R_0 \cup E_0$ is used to decide of the AC theory for some symbol f. The Theorem 5.1 states that any equation $s = t$, not part of the system $R_0 \cup E_0$, where $s =_{AC(f)} t$, is redundant and can be disposed of during the proof search process. Let us examine how this theorem is proved:

$R_0(E_0)$ (the set of orientable instances of the equations) is terminating by construction. For every critical pair, all of its ground instances are joinable. Hence $R_0(E_0)$ is confluent on $\mathrm{Term}(F^e)$. Consequently, if $s =_{R_0 \cup E_0}$, then $s\sigma \downarrow t\sigma$ for any ground substitution σ, and therefore $s \Downarrow^{\triangleright} t$.

We can adapt this proof, for a given reduction ordering $\succ$, to any set of equations E that is ground convergent. Indeed, with the trivial rewriting system $R = \emptyset$, $R(E)$ is ground convergent and terminating (included in the well-founded reduction ordering $\succ$). Let us consider an equation $s = t$, and write $s' = t'$ the same equation where variables $x_0, \ldots, x_n$ are replaced by fresh constants $c_0, \ldots, c_n$; let us extend the ordering $\succ$ to a reduction ordering $\succ'$ that contains $\succ$ ([2] explains how to do it for LPO and KBO, respectively, in lemmas 5.2 and 5.3). Then, if $s' \downarrow^{R(E)^{\succ'}} t'$, every ground instance of $s = t$ is joinable by $E^{\succ}$, and $s = t$ is redundant.

In other words, E provides us with a redundancy criterion for any equation $s = t$, by checking whether s' and t' have the same normal form. In practice, we just have to consider variables in s and t as constants, extend $\succ$ with those new constants, and compute the normal form of both terms w.r.t. orientable equations of E. The resulting simplification rules are exposed in Figure 4. The

Tautology Deletion modulo $\mathcal{T}$:

$$\frac{s = t \vee C}{\;} \text{ if } \text{const}(s) \downarrow^{\mathcal{T}} \text{const}(t)$$

Equality Resolution modulo $\mathcal{T}$:

$$\frac{s \neq t \vee C}{C} \text{ if } \text{const}(s) \downarrow^{\mathcal{T}} \text{const}(t)$$

Figure 4. Simplification Rules for the Redundancy Criterion on a Theory $\mathcal{T}$

double bar indicates that the clause above is *replaced* by the clause below. The operation const replaces variables in s and t by fresh constants; ground joinability of s and t is then implied by joinability of const(s) and const(t).

Theory combination If several theories $T_1, \ldots, T_n$ occur in a single problem, then we can combine several ground confluent systems $E_1, \ldots, E_n$. The combination will always be terminating (because included in $\succ$), but not necessarily ground-convergent any more. However, if the theories have disjoint signatures, the combination is still a decision procedure on terms that are exclusively composed of free symbols and symbols from T_i for some i. On mixed term, we may want to purify terms by introducing fresh constants for subterms that belong to a different theory.

Interaction with Datalog Now that we have a redundancy criterion for some theories, we can encode it, regardless of the concrete signature, into Datalog clauses. The encoding is more complicated than previous ones — it involves boxing several patterns, keeping track of a relationship between Datalog variables and the symbols of each equation of a ground joinable system — but it follows the same principles. Then, such a redundancy criterion can be triggered when the theory it decides is detected; the equational theorem prover can then use it, if its ordering is compatible. The E[11] prover does exactly this, for the specific case of AC symbols (with the same rules as in Figure 4 where $\mathcal{T}$ is replaced by AC for a set of symbols).

Computing Criteria for a Theory If we want to compute such a ground convergent system of equations E for a given theory E_0 (for instance, AC or a formulation of Group theory), a possibility is to use the *syntactic criterion* described in Theorem 5.2 of [2] in order to saturate E_0 (in a given ordering $\succ$), while discarding ground joinable equations. If this process terminates, it yields a set E that must be ground convergent (otherwise there would be a non ground-joinable equation in E).

Given an equational theory E_0 that has symbols $f_1, \ldots, f_n$, we can try to compute such ground convergent systems by saturation for a finite set of LPO and KBO orderings on those symbols. Whenever a saturation succeeds for some ordering $\succ$, it yields a decision procedure E for E_0 in $\succ$. If we later meet a problem where the axioms contain E_0, and the ordering is compatible with $\succ$, we add E to the set of clauses and remove any clause $c \notin E$ that is ground-joinable by $E^{\succ}$.

5.3 Term Orderings

In [6], the authors describe a system, quite similar to ours, used by Waldmeister during the preprocessing phase to detect some theories and heuristically choose a term ordering that experience has shown to be efficient for those theories. This kind of analysis of the problem is feasible with our approach as well. However, since theories can be detected during the proof search, information on the ordering may come too late; in this case, *restarting* the prover with a different ordering, chosen with more information about the problem — maybe keeping some useful deduced clauses, like rewriting rules — can be relevant.

6 Experimental Results

We compared our experimental implementation[4] (version 0.2) with SPASS[14] and E[11] on categories RNG and GRP of the TPTP[13] base of problems. Benchmarks include both `zipperposition` — our theorem prover with theory detection, relational un-mangling lemma, and redundancy criteria (for AC, commutative monoids and abelian groups) — `zipperposition-lemma`, with the relational lemma but no redundancy criteria, and `zipperposition-no-theories`, in which all theory handling is disabled. The results are exposed in Figure 5. Overall, on 1434 problems, `zipperposition` proves 7 problems that are not proven by SPASS nor E within the 120s timeout. Zipperposition is able to detect at least one theory in 594 problems out of 1434, and triggers the lemma in 68 problems. Among the 594 problems with theories, 31 are solved by *zipperposition* or *zipperposition-lemma*, but not by *zipperposition-no-theories*, and 7 are solved by the latter but not by the former (because the prover was slower or it pruned the wrong part of the search space). This ratio becomes 7 to 2 on the problems in which the lemma is applied.

We can already see that the redundancy criterion, with its quite naive implementation, already brings benefits. The un-mangling lemma makes a significant difference on the set of problems in which it applies. On individual problems, the difference can be striking: some problems that would not terminate within 2 minutes become trivial enough to get solved in 0.5s when lemma detection is enabled. Those results are encouraging, and we believe that using a meta-prover

[4] We point out that our implementation of superposition is not nearly as good as SPASS or E, which are the result of years of work.

Prover	Proved	success rate(%)	proved /594	%	proved /68	%
E	1047	73.0	430	72.4	59	86
SPASS	863	60.1	376	63.3	50	73
zipperposition	531	37.0	202	34.0	56	82
zipperposition-lemma	527	36.7	199	33.5	57	83
zipperposition-no-theories	504	35.1	191	32.1	52	76

Figure 5. Benchmark Results: Number of Solved Problems

may find more uses in automated theorem proving. Profiling shows that the Datalog reasoner represents a negligible fraction of the run-time (less than 1%). On the other hand, our implementation is more naive and less efficient than SPASS or E (which have a more powerful calculus, better heuristics, or a more efficient implementation), which can explain why they still solve more problems. Our technique could be integrated in other theorem provers to discover lemmas or usable redundancy criteria — especially for scheduling provers (like iProver[7]) because meta-level facts that are discovered during a time slice can be used for the next ones (using a suitable term ordering, etc.).

Three problems are solved only by the versions of Zipperposition that use lemma detection: GRP392-1.p, GRP393-1.p and GRP394-1.p. Interestingly, all three are satisfiable problems in relational form where the un-mangling lemma transforms into easily saturated sets of equations. This is only possible because the calculus of [5] turns some equivalences into rewrite rules.

Conclusion and Possible Extensions

We have shown a generic and flexible way to detect instances of axioms and theories during the search for a (clausal) proof. The use of a Datalog incremental inference system, which manipulates assertions about the problem itself, makes the meta-level reasoning flexible, modular and allows to have one's own meta-facts (properties) triggered by new meta-assertions. This technique already shows very promising results, and can be improved further with more sophisticated uses of the detected theories. We believe that this kind of combination, although still quite simple, bears some resemblance with the way real mathematicians solve problems. Using several levels of description and proof may also help making automated proofs more understandable, saturation proofs being often blamed for being very unintuitive to human users. Further development includes:

- making the reasoner more proactive by having it spawning subprocesses to try to prove missing axioms;
- computing redundancy criteria for equational theories off-line. Theories could be extracted from axiom files, before a redundancy criterion is looked for by saturating the axioms;

– automatically extract lemma from successful proofs in order to help solving similar problems;
– implementing this technique in a state of the art prover.

Acknowledgements

We would like to thank Gilles Dowek for his help, and the anonymous reviewers for their detailed and helpful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Avenhaus, J., Hillenbrand, T., Löchner, B.: On using ground joinable equations in equational theorem proving. Journal of Symbolic Computation **36**(12) (2003) 217 – 233
3. Bachmair, L., Ganzinger, H.: Associative-commutative superposition. In Dershowitz, N., Lindenstrauss, N., eds.: Conditional and Typed Rewriting Systems. Volume 968 of Lecture Notes in Computer Science. Springer (1995)
4. Denzinger, J., Schulz, S.: Learning domain knowledge to improve theorem proving. In McRobbie, M., Slaney, J., eds.: Automated Deduction Cade-13. Volume 1104 of Lecture Notes in Computer Science. Springer (1996) 62–76
5. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. In Baader, F., ed.: Automated Deduction CADE-19. Volume 2741 of Lecture Notes in Computer Science. Springer (2003) 335–349
6. Hillenbrand, T., Jaeger, A., Löchner, B.: System description: Waldmeister improvements in performance and ease of use. In: Automated Deduction CADE-16. Volume 1632 of Lecture Notes in Computer Science. Springer (1999) 232–236
7. Korovin, K.: iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: Proceedings of the 4th international joint conference on Automated Reasoning. IJCAR '08, Berlin, Heidelberg, Springer-Verlag (2008) 292–298
8. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (1999)
9. Riazanov, A., Voronkov, A.: Vampire 1.1 (system description). In: Proceedings of the First International Joint Conference on Automated Reasoning. IJCAR '01, London, UK, UK, Springer-Verlag (2001) 376–380
10. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM **12**(1) (1965) 23–41
11. Schulz, S.: E - a brainiac theorem prover. AI Commun. **15**(2,3) (August 2002) 111–126
12. Stuber, J.: Superposition theorem proving for abelian groups represented as integer modules. In: Theoretical Computer Science, Springer (1996) 208–1
13. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning **43**(4) (2009) 337–362
14. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System Description: SPASS Version 3.0. In Pfenning, F., ed.: Automated Deduction CADE-21. Volume 4603 of Lecture Notes in Computer Science. Springer (2007) 514–520